Towards a Formally Verified Kernel Module

Joaquim Tojal[1,2], Carlos Carloto[1], José Miguel Faria[2], and Simão Melo de Sousa[1,3⋆]

[1] University of Beira Interior,Dept. of Computer Science, Covilhã, Portugal
[2] Critical Software SA.,Coimbra, Portugal
[3] University of Porto, Artificial Intelligence and Computer Science Laboratory (LIACC-UP), Porto, Portugal

Abstract. In this article we present the design by contract approach to formal verification of an industrial real-time kernel using VCC (Verified C Compiler) and Frama-C tools. The annotations were directly inserted into the source code of an industrial kernel module, xLuna, and verified automatically. VCC was also used to reason about concurrency issues in a preemptable and real-time environment. In addition we describe some particular methodological aspects of these two verifiers. These are the first results towards a Formally Verified Kernel.

1 Introduction

Almost every computer system depends directly on the operating systems behavior. As such, having kernel code that is proved to be correct is a goal that researchers and industrial companies have attempted to achieve. Large amounts of low-level implementations like operating system core is obviously a perfect and challenging target for formal verification. The interest in formally verifying realistic and industrial low-level code and obtaining the highest standards of safety like Common Criteria EAL7 has grown significantly in recent years. In this work we target precisely this goal taking a modular approach to formally verify a real industrial real-time operating system kernel which was not designed with formal verification in mind. The kernel targeted is a particular interrupt manager of xLuna real-time kernel for embedded systems built by Critical Software,SA. For verification, we use Microsoft Research Verified C Compiler(VCC) and Frama-C tools to reason about functional correctness, concurrency and safety properties of xLuna kernel. The specifications are based in Hoare-style pre- and post-conditions inlined with the real code.

The remaining of this paper is organized as follows: Section 2 show the xLuna architecture and some particular design aspects. Then, in section 3, we gave an overview toolchain and verification methodologies for both, VCC and Frama-C. Section 4 exposes the detail design of xLuna IRQ manager and respective verification approaches. At the end, sections 5 and 6 give some conclusions that led us to future improvements and related work to the subject.

⋆ This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI and the project RESCUE (PTDC/EIA/65862/2006)

2 xLuna

xLuna [17] is a microkernel based on the RTEMS [18] Real-Time Operating System with the ability to run a GNU/Linux Operating System [19], providing therefore a runtime environment for real-time (RTEMS) and non-real-time (Linux) applications. xLuna was designed to supports ESA's LEON SPARC processor [17] with the main goal of extending the RTEMS kernel in order to enable a safely Linux execution without jeopardizing aspects of reliability, availability, maintainability and safety. Its general architecture is shown in Figure 1. The

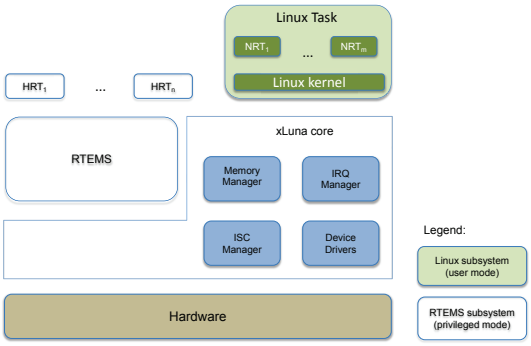

Fig. 1. xLuna architecture

Linux subsystem runs as an unprivileged RTEMS task and in a different memory partition. This provides spacial partitioning, guaranteeing a safe isolation between the non-real-time (NRT) and hard-real-time (HRT) subsystems. xLuna provides four main modules:

- *Memory manager*: enforces the isolation requirements between RTEMS and Linux and memory protection of Linux kernel from NRT user processes.
- *Interrupt Manager (IRQ)*: connects interrupts of the real hardware to the Linux kernel (which does not have access to them) and provides services.
- *Inter-System communication (ISC)*: for bidirectional communication between HRT and NRT tasks.
- *Device Drivers*: for other hardware virtualizations required (e.g. timer).

In the following section we describe the important aspects of the verification tools used on xLuna's kernel.

3 Methodology and Tools

3.1 Verified C Compiler (VCC)

VCC [11] is an automatic verification tool for concurrent C that is being developed by Microsoft Research, Redmond, USA and European Microsoft In-

novation Center (EMIC), Aachen, Germany. VCC utilizes annotations based in Hoare-style pre- and post-conditions closely attached to source code. That is, the annotations are mixed into the *codebase* rather than within blocks of commented (for the C compiler) specifications. Even so, the annotated C code can still be compiled with any C compiler through conditional compilation: If a regular C compiler is called, a special VCC flag is disabled and the annotations and specification code are preprocessed and transformed into empty strings. The concept of complete code with annotations is the same used in JML [16] for Java code or SPEC# [13] for C# programs.

Verification Toolchain VCC is fully integrated with the well known Microsoft Visual Studio IDE, providing a familiar environment to programmers and making the correct use of the VCC toolchain very simple. Three main steps are made by VCC when trying to verify an annotated C program: (i) VCC translates annotated C code into BoogiePL (an intermediated verification language), (ii) Boogie translates BoogiePL into first-order predicate formulas (verification conditions) and (iii) Z3 tries to solve them and if it finds a proof then the program is correct according to the specifications.

VCC Annotations As in standard *design by contract* approach to partial correctness, the pre- and post-conditions inserted into a C function constitute a contract between the function and a function caller, guaranteeing that if the function starts in a state that satisfies the precondition then the postcondition holds at the end of the function. To express and reason about specifications, VCC uses clauses such *requires*, *ensures*, *result* or *writes* representing, respectively, pre- and post-conditions, function return value, and frame conditions which limit what the function is authorized to modify. VCC also supports loop invariants for reasoning about loop behavior and termination.

Ownership, Type Invariants and Ghost Code VCC memory model ensures that objects (pointers to structures) do not overlap in memory keeping a typed and hierarchical view of all objects (Spec# ownership model). This ownership machinery is applied as a *ghost* transformation behind every structure in the program. Each type has a related ownership control object in specification code (*ghost* code), only visible to VCC and providing a bridge between implicit specification code and VCC annotations in the real program. For the sake of brevity the transformation process is not detailed in this paper (a more detailed account can be found in [21], available at `http://www.di.ubi.pt/~release`). *Ghost* code can also exist as explicit specification functions, objects or variables only seen by VCC for verification purposes. Structures can be also annotated with invariants related to their ownership behavior or to their own fields.

VCC implicitly lets an object own its representation and writing the object allows writing its ownership domain. However, updating an object requires a special *wrap/unwrap* protocol to transfer ownership domains (see Figure 2).

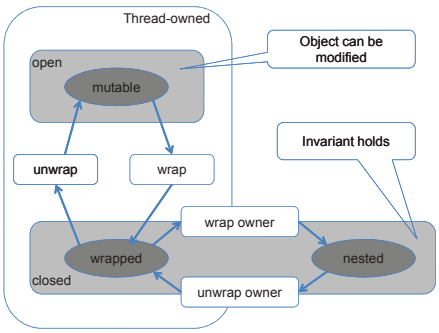

Fig. 2. VCC wrap/unwrap protocol

Defining an hierarchical structure gives VCC a tree view of the system. One object can only have one owner and threads own themselves. Figure 2 shows that in a specific ownership domain each object should be *open* or *close* inside of the thread domain and *close* outside the thread domain. In a sequential fashion an object can only be updated when it is *mutable* and must return to a safe state performing some operations required by the *wrap*/*unwrap* protocol. The explanation of these states is given bellow.

- **Mutable**. After creation the object is open and owned by *me*() which represents the current thread. In this state the thread is allowed to modify the object and prevents other threads interference.
- **Wrapped**. Wrapping the object (*wrap*) requires its invariant to hold. The reverse transition (*unwrap*) assume object invariants.
- **Nested**. Closed objects can be added to (or removed from) another objects ownership via *set_owns*(), *set_closed_owner*() (wrap owner) or unwrap owner operations using *giveup_closed_owner*(). The reverse transition (*unwrap*) assume object invariants.

3.2 Frama-C

Frama-C is a platform dedicated to the analysis of software source code written in C. It gathers a series of different tools with several static analyses techniques and a deductive verifier in a single collaborative framework. Its collaborative approach allows different analyzers to build upon the results previously computed by other analyzers. Frama-C is an extensible framework. It is open source software and is organized with a plugin architecture. It contains several ready-to-use plugins and new plugins may be built and use the results or functionalities provided by the existing plugins. A common kernel centralizes information and conducts the analysis. Plugins interact with each other through interfaces defined by the kernel.
Currently, Frama-C provides several *lightweight* analyzers (e.g., "Metrics", "Call-graphs", "Users", "Constant Folding", and "Occurrence"), semantic analyzers

(e.g., “Functional Dependencies”, “Slicing”, and “Impact”), the sophisticated “Value Analysis” plugin, which automatically computes variation domains for the variables of a program, and deductive verification, through the “Jessie” plugin. Jessie is the Frama-C plugin that enables design by contract development of C programs. The contracts are written in the ANSI/ISO C Specification Language (ACSL) [15]. The generated verification conditions can be submitted to external automatic provers such as Simplify, Alt- Ergo, Z3, Yices, and CVC3.

4 Verification of xLuna IRQ Manager

Verifying low-level code that was not implemented with formal verification in mind and adapting the verification methodology to that implementation is a non-trivial task. When reasoning about xLuna as a formal verification target its own architecture suggests that a modular approach should be taken in order to achieve an overall correctness proof. One of the most critical and crucial modules of xLuna is the IRQ manager, since it has to catch and process all interrupt requests needed for proper system functionality. It thus constitutes an interesting target for formal verification and was the subject of our study. Yet, the different modules are considerably dependent on each other. Moreover, kernel code is highly dependent of machine assembly instructions, which causes issues for the verification task since VCC does not interpret assembly language. For this reason, it was decided, at this stage, to assume that machine instructions and inlined assembly are correct. All IRQ module dependencies were studied to build proper code isolations and abstractions were made to fit the verification methodology.

4.1 IRQ Design

Following xLuna architecture, the Linux kernel is running as an unprivileged (user-mode) RTEMS task and thus, it does not have direct hardware access. The main purpose of the IRQ manager is to serve as a bridge connecting the Linux subsystem to hardware interrupts. This enables catching incoming hardware interrupts required by Linux kernel, system calls made by Linux processes, or xLuna system calls made by Linux kernel. Hardware interrupts or software traps are both called *events* that are inserted into an *Event Queue* to be sent to their handlers.
Interrupts are filtered by a dispatcher function which is responsible to insert them into the event queue. Once there are events in the queue they should be processed by the IRQ manager through a *monitor task* which calls the respective Linux handlers or xLuna services (see Figure 3). Events can be treated synchronously or asynchronously. In the queue structure there can be only one *sync* event at a time. This event is a request caused by the running instruction and must be processed synchronously. *Async* events are accumulated in the queue according to their event type (e.g., TT_ISC_RTEMS_TO_LX is the special trap type 0x21 for inter systems communication that is inserted as an async event). As shown in

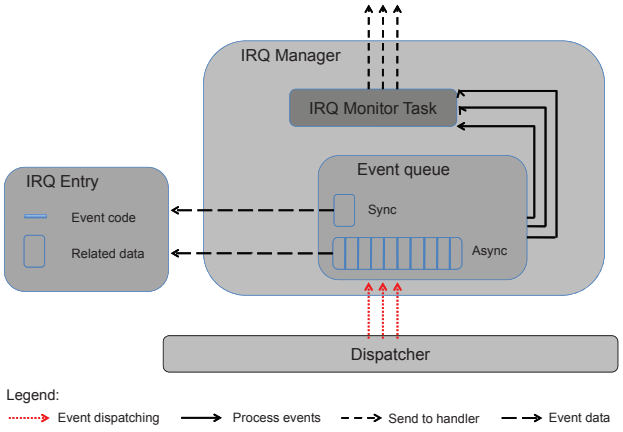

Fig. 3. Interrupt request manager

Figure 3, both *Sync* and *Async* events have an associated data structure which contains the data needed for event handlers and the interrupt/trap code.

In the next sections, we describe the approach taken for verifying the treatment of each event inserted in the queue and the correct usage of data structures evolved in this flow.

4.2 VCC Verification Approach

We can drive the verification methodology used, through the analysis of an IRQ manager function used to insert a synchronous event into the queue. As already mentioned, VCC enforces a ownership model to guarantee a consistent and typed view of all objects (e.g., pointers to data structures) which ensures for instance that objects of the same type and different addresses do not overlap in memory [12].

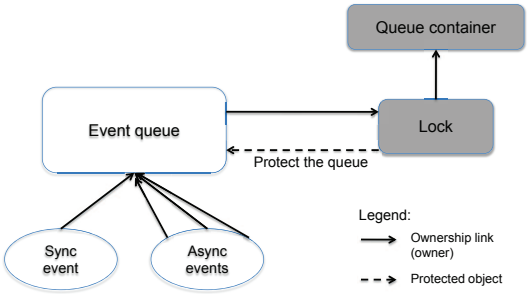

Fig. 4. VCC event queue ownership

When xLuna interrupt manager is initialized, VCC considers all objects as mutable. Therefore ownership relations must to be configured at the program entry point (*irq_init()* function) to prevent further object updates without complying with the ownership protocol. This setup ensures at IRQ manager boot time that:

- The queue is *typed* and protected at the end of the initialization;
- The queue is the owner of all *sync* and *async* events;
- When the queue is *closed* its invariant and the invariants of all embedded types hold.

With this ownership setup in mind, the following example illustrates the addition of a *sync* event to the queue. The code is a simplified version of the original function.

```
1  int event_queue_insert_sync(struct irq_entry *irq claimp(c))
2    always(c,closed(&QueueLock))
3    requires(nested(&event_queue))
4    requires(irq->data != NULL)
5    requires(!VCC_SPEC_FUN_is_asyn_trap(irq->event))
6    requires(thread_local(irq))
7    reads(irq)
8    writes(&event_queue)
9    ensures(&event_queue.sync_event == irq)
10   ensures(nested(&event_queue))
11   {
12   //xLuna assert function
13   ASSERT(irq->event);
14     assert(typed(&irq->event));
15   sparc_disable_interrupts();
16     spec(VCC_SPEC_FUN_sparc_disable_interrupts(spec(&QueueLock
         ) spec(c));)
17     // unwrap/wrap protocol
18     unwrap(&event_queue);
19     unwrap(&event_queue.sync_event);
20   //insert sync event(for vcc we can now change the queue)
21   event_queue.sync_event = *irq;
22     //reverse wrapping
23     wrap(&event_queue.sync_event);
24     wrap(&event_queue);
25   sparc_enable_interrupts();
26     spec(VCC_SPEC_FUN_sparc_enable_interrupts(spec(&QueueLock)
         spec(c));)
27   return 0;
28 }
```

Listing 1.1. Insert synchronous event function in VCC

As preconditions one tells to VCC that this function will start in a state where:

- *event_queue* is owned by an object and thus its invariant holds (line 3);

– *sync* events always have *data* required by event handlers to work properly (line 4);
– event code must be within sync event bounds (line 5);
– *irq* points to an object that is local to the running thread (lines 6) which is only allowed to read the same *irq* object (lines 7). On the other hand, permissions to change *event_queue* are necessary. The *writes* clause guarantees that only *event_queue* will be updated (line 8).

To respect the VCC ownership protocol one needs to *unwrap* (lines 18, 19) the queue and its embedded IRQ entry, change it and then *wrap* in reverse order (lines 23, 24). When an object is *unwrapped*, VCC implicitly *assume* its invariant and *assert* it when wrap occurs. In the example, this ownership flow will guarantee the second postcondition (line 10), whereas the first one ensures that the queue *sync* event was updated correctly (line 9).

Lines 2, 15, 16, 25, 26 and the ghost claim parameter (line 1) are related with concurrency verification and are explained in next section.

4.3 Concurrency Verification

The above described approach is suitable for sequential code. Nevertheless, the kernel has to guarantee that executions are made without interference or unexpected kernel behavior. In a concurrent real-time kernel, not only user processes are preemptable but also kernel processes may be subject to stringent scheduling policies or interrupts. In xLuna's implementation, in order to achieve non-interference updates of critical regions, low-level functions (implemented in assembly language) that disable and enable back interrupts are used. The required update steps are performed while interrupts are disabled. In our verification approach, such assembly functions are represented by VCC ghost code instructions.

Lock approach Disabling interrupts gives to the running thread non-interference execution steps before interrupts are enabled again. One can think about this low-level access policy as a mandatory lock and use an abstract ghost implementation suitable for VCC methodology. When interrupts are disabled we call a VCC ghost specification function (see Listing 1.1 line 16) that will transfer ownership domain to the running thread and prevent preemption while interrupts are enable (line 26). This allows reasoning about implementation steps in a sequential fashion. Still, one needs to ensures that the lock is not destroyed or deallocated: in VCC we can model this issue through a *claim*. The gray part of Figure 4 shows the ownership configuration for ghost code and states that the *Queue Container* ghost structure is the owner of the lock and the latter owns the queue. The *container* has two invariants saying that (i) the object protected by the lock is the *queue* and (ii) it is the owner of the lock. In VCC this knowledge can be passed to functions through ghost parameters (claims). In the example, the ghost claim parameter and the clause in line 2 guarantee that:

– c has an implicit invariant ensuring that the *container* remains *closed* (*container* invariant holds)
– the *always* clause tells VCC to enforce the fact that if the *container* is *closed*, the lock is also *closed* and thus it can never be destroyed.

4.4 Frama-C Verification Approach

The main focus of the Frama-C verification was the safety and functional issues of xLuna IRQ Manager. Before that, the initial stage comprised the evaluation and tailoring of the original code: since, at the moment, Frama-C does not support all the C language, some functions had to be changed to resolve some incompatibilities. The main topics covered were:

– *Pointer dereferencing*: the code of the xLuna IRQ Manager has a significant number of global variables and pointers. As such, pointer dereferencing was one of the most relevant aspects;
– *Arguments*: analyze if the function arguments are correctly introduced and do not prevent the functions from terminating;
– *Loops*: analyze the body of the loops and build the necessary contracts (loop variants and loop invariants) to prove that each loop terminates

To illustrate the Frama-C verification approach, we consider the same example, i.e., the function that inserts synchronous events into the event queue.

```
1  /*@ requires \valid(irq);
2    requires (irq->data!=NULL);
3    ensures \result==0 || \result==(-1);
4      @*/
5  int event_queue_insert_sync(struct irq_entry *irq)
6  {
7    int level;
8    ASSERT(irq->event);
9    if (event_queue_has_sync_event()) return -1;
10   //@ assert \valid(irq);
11   //@ ensures event_queue.sync_event == *irq;
12   event_queue.sync_event = *irq;
13   return 0;
14 }
```

Listing 1.2. Insert synchronous event function in Frama-C

The contracts included in this function are explained as follows:

– Line 1: This contract is a precondition: in order to the function finishes it needs a valid irq. It is required that irq pointer is allocated in a safely memory location.

– Line 2: Also a precondition. In this contract we say that the data cannot be null.
– Line 3: A postcondition that guarantees that the output of the function is either 0 or -1. (It is 0 if the event is inserted; if not the output is -1.)
– Line 8: This ASSERT is not part of the contracts that we build. It is an original xLuna C statement.
– Line 10: An assertion. It strengthens the precondition on line 1. It is right before the place where is absolutely necessary that irq is not NULL.
– Line 11: The postcondition of the function.

For this function, it were generated 15 proof obligations; all of them proved with success by the automatic prover.

The proof obligations were generated by why, and Alt-ergo was the automatic prover utilized to discharge them. In the total of the project, it were generated 373 proof obligations., all of them automatically proved by Alt-ergo. The results are analyzed in the next section.

5 Conclusions and Future Work

In this paper we have presented the design by contract approach to formal verification of a realistic system using different verification tools for a feasibility study. One can instantly conclude that the annotations burden in VCC is greater than Frama-C, mainly due to the ghost code and type invariants supported by VCC. In the overall IRQ model (about 1k LOC C) were inserted almost ten times more VCC annotations than Frama-C. Safety properties such as correct array index, arithmetic overflow and pointer deference or null pointers were verified in most parts of the IRQ manager. All inside function updates were surrounded by frame conditions and all parameters validated, type invariants hold with respect to the VCC ownership protocol. As said before VCC memory model guarantees that objects do not overlap in memory. At this time, pre- and post-conditions were added to approximately 80% of IRQ manager C code. The rest of the IRQ C code is related to switching and Linux stack manipulations. The concurrency verification approach guarantees sequential execution in a preemptable environment and it is proved to be suitable to VCC methodology [20,10] and also used in PikeOS. However, assembly language in kernel code can not be ignored when aiming to an overall system verification. One possibility to extend VCC work is the construction of a ghost model of the underlying hardware and assembly language, and connect the specifications made to the ghost model.

After this work we are able to do an analysis of the platform Frama-C. It is a platform that is being developed and in the last months it has been in constant evolution. In this moment it is a platform that is good to work with. The integration of the ACSL, Jessie and the why platform makes Frama-C a very good platform to work in the verification of programs. However it has some problems that the developer needs to improve. Some of those problems are:

– Frama-C has limited support for function pointers. This is a problem when used on the type of code found in an operating system module.

– It currently lacks clear error messages describing what and where the problem is.
– Its implementation is not stable yet (it is an academic tool), it crashes often. This mostly happens when there are pointers present in the code to be verified.
– Some annotations cause crashes of Frama-C or its subsystems.

With Frama-C, in the functions that we analyze, the results on the automatic prover was a success, and all the proof obligations of those function were checked with success. As said before, some code of a few functions were removed because of Frama-c incompatibilities, and other functions weren't analyze because of completely incompatibility with Frama-C. We have verified about 80% of the IRQ Manager code. So as the Frama-C development proceed it will be possible to reintroduce the code removed and to analyze the functions that we couldn't analyze with the actual version of Frama-C. After the work done in the verification of the xLuna IRQ Manager we should be able to proceed to other xLuna modules. As long as the xLuna modules aren't too incompatible with the actual version of Frama-C the verification of those modules may be possible. However with the evolution of Frama-C it may be possible to verificate all of xLuna modules.

6 Related Work

Prove correctness of low-level software implementations is a goal pursued for decades, the early work on formal verification of operating systems comes from 1973-1980 with the Provably Secure Operating System (PSOS) [1]. Later in seventies UCLA Data Secure Unix (DSU) [2] was the first approximating the modern microkernel architecture. The proofs were guided based on first-order predicate calculus and 20% of the code have been proven correct. Other early work is the Kernel for Isolated Tasks (KIT) [3], KIT address the problem of verifying properties for process isolation in a multi-tasking environment. PSOS, DSU and KIT were pioneers in attempts to large scale software verifications and inspired some techniques still used nowadays. Verified Fiasco (VFiasco) [4] project started in 2001 with a experiment using SPIN model checker to verify a small version of the Fiasco inter-process communication. After this experiment the project moved on using the PVS theorem prover to formalize a subset of C++ to reason about Fiasco implementations. SPIN model checker was also used in other kernels such as Fluke [5], RUBIS [6] or HARMONY [7]. The L4 micro-kernel has also been a target for formal verification projects, the most recent in seL4 project [8]. seL4 was concluded at the end of 2007 with a resulting small (8700 LOC C and 600 LOC assembly) microkernel for run in ARM architecture. The OS design team used Haskell and Isabelle/HOL for fast prototyping and specification proofs. More within the scope of this paper are the VerisoftXT project which uses VCC verification methodology to prove correctness of Microsoft Hyper-V Hypervisor [9] and SYSGO PikeOS microkernel [10].

References

1. P. Neumann, R. Feiertage: PSOS Revisited. Proceedings of the 19th Annual Computer Security Applications Conference (2003) 208
2. B. Walker, R. Kemmerer, G. Popek: Specification and verification of the ucla unix security kernel. Commun. ACM (1980) 118-131
3. W. Bevier: A verified operating system kernel. Report 11, Computational Logic Inc., Austin, Texas (1987)
4. M. Hohmuth, H. Tews: The vfiasco approach for a verified operating system. In 2nd ECOOP Workshop on Program Languages and Operating Systems (2005)
5. P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, G. Back: Formal methods: A practical tool for os implementors. Proceedings of the 6th Workshop on Hot Topics in Operating Systems (1997)
6. G. Duval, J. Julliand: Modeling and verification of the rubis microkernel with spin. In Proceedings of the First SPIN Workshop (1995)
7. T. Cattel: Modelization and verification of a multiprocessor real-time OS kernel. Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII (1995)
8. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood: seL4: Formal verification of an OS kernel. In Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP) (2009) 207-220
9. D. Leinenbach, T. Santen: Verifying the Microsoft Hyper-V Hypervisor with VCC. Refinement Based Methods for the Construction of Dependable Systems, number 09381 in Dagstuhl Seminar Proceedings (2010) 104-108.
10. C. Baumann, B. Beckert, H. Blasum, T. Bormer: Ingredients of Operating System Correctness. Embedded World 2010 Conference (2010)
11. E. Cohen , M. Dahlweid , M. Hillebrand , D. Leinenbach , M. Moskal , T. Santen , W. Schulte , S. Tobies: VCC: A Practical System for Verifying Concurrent C. Theorem Proving in Higher Order Logics (2009)
12. E. Cohen, M. Moskal, W. Schulte, S. Tobies: A Precise Yet Efficient Memory Model for C. 4th International Workshop on Systems Software Verification (2009)
13. M. Barnett, K. Leino, and W. Schulte: The Spec# programming system: An overview. In CASSIS 2004, LNCS vol. 3362, Springer (2004)
14. Frama-C Web page: `http://frama-c.com/`
15. J. Burghardt, J. Gerlach, K. Hartig, J. Soto, C. Weber: ACSL By Example: Towards a Verified C Standard Library (2010)
16. G. Leavens, and Y. Cheon: Design by Contract with JML. JML tutorial (2006)
17. P. Braga, L. Henriques, M. Zulianello: xLuna:eXtending free/open-source reaL-time execUtive for oN-bord space Applications. Small Satellites Systems and Services The ESA 4S Symposium (2008)
18. RTEMS web page: `http://www.rtems.com/`.
19. Snapgear Embedded Linux web page: `http://www.snapgear.org/`.
20. E. Hillebrand, D. Leinenbach: Formal Verification of a Reader-Writer Lock Implementation in C. 4th International Workshop on Systems Software Verification (2009) 123-141
21. J. Tojal. Towards a Formally Verified Microkernel using the VCC Verifier. Master's thesis, University of Beira Interior, Portugal (2010)